On Measurement and Analysis of Software Changes

Audris Mockus, Stephen G. Eick, Todd L. Graves, Alan F. Karr

Abstract

Software becomes better or worse because of the changes made to it. Each change to legacy software is expensive and risky but it also has potential for generating revenues because of desired new functionality or cost savings in future maintenance. Hence it is important to understand and quantify change properties in order to make good business decisions. Particularly good sources of historical information about changes, especially for legacy software, are generated by change management systems used by most large software organizations. This paper describes how to measure change properties and how to use that information to identify cost and quality drivers in software production. The methodology is codified in a system called *SoftChange*. *SoftChange* was developed for use with the 5ESS$^{\text{TM}}$switch at Lucent Technologies, but can also be used with other software projects using the same, widespread version control systems. *SoftChange* defines, constructs and presents essential measures of software change size, complexity, and developer expertise. Also, it provides tools for imputing the purpose and effort required for a change, and generates predictions of the quality of a change. This methodology uses measurements of changes to benefit the software development process and to generate insights about software evolution.

A. Mockus is with Bell Laboratories.
S. G. Eick is with Bell Laboratories.
T. L. Graves is with the National Institute of Statistical Sciences and Bell Laboratories.
A. F. Karr is with the National Institute of Statistical Sciences.

I. Introduction

Legacy software development organizations are constantly under pressure from new hardware and the changing business environment to modify their software. However, changes to the software are hard to accomplish because of the software's age and size and the limited number of expert developers available. Despite the landmark studies of software evolution started decades ago [1] there is little research on measurement and quantitative analysis of software changes. This is in stark contrast to extensive literature that exists on source code complexity [2], [3], complexity of an object oriented design [4], or functional complexity [5].

To analyze large-scale software production it is essential to have extensive and reliable sources of information. An underutilized and rich source of information is the data generated automatically by the version control and change management systems. This paper will illustrate how to make use of change management data to measure properties of software changes and how such measures can be used in making inference about cost and quality drivers in software production.

To automate the change measurement and analysis process, we have built a collection of tools which we refer to as *SoftChange*. A great variety of studies have depended on the measures of change properties embodied in *SoftChange*. We developed the tools while studying the problem of code decay [6] in the context of the 5ESS$^{\text{TM}}$, a large legacy switching software system. Change properties measured via *SoftChange* help demonstrate the existence of code decay [6]. Together with interviews of key personnel, data from *SoftChange* were instrumental in studying the interdependencies of software development and the organizational structure in legacy organizations [7]. Change properties can also be used to quantify increases in fault incidence due to past changes [8] or the increased complexity of parallel changes [9]. *SoftChange* includes methodology for reliably inferring the purpose of changes [10], and an iterative algorithm described in [11], [12] for identifying factors that affect the effort required for individual changes. The presence of *SoftChange* has made it quite straightforward to perform analyses using measurements on changes, so we expect other advances in the understanding of software production to follow.

The main sources of information for the *SoftChange* system are version control and configuration management databases. The way these data are stored and collected tends to be uniform over time and for different parts of the software. In addition, most large software projects use similar databases. It is important to note that the version control systems are used to allow tagging of different versions of the code and to help large teams of developers to work on the same code base. Change data are collected as a side effect when version control systems keep track of versions. Other methods of collecting information make use of surveys or experiments [13], [14]. The resulting information is valuable and detailed, but limited in scope and time, even if the data gathering is not prohibitively expensive. Further, the very act of performing the experiment can alter developers' behavior and thereby bias the results. *SoftChange* is not susceptible to these difficulties since it makes use of data gathered automatically as part of normal processes.

The change measures provide a new information infrastructure for managers, as well as for future research on large software systems. The new insights generated by using measures of software change underline the importance of studying changes to the source code. The *SoftChange* system codifies the knowledge about large scale commercially successful software that can be and currently is being transferred to other software products.

5ESS™, a sixteen-year-old real-time software product for telephone switches, was the initial focus of our investigation. Currently the product comprises 100,000,000 lines of source code, has more than 50 major subsystems with more than 10,000 modules and more than 350,000 files.[1] The source code has been changed well over four million times. The sheer size of the version management database precludes unsupported analysis. To provide comprehensive information about the entire software product, over its entire lifetime, a system for change history data analysis must be able to handle this size, in addition to confronting the following difficulties. First, some important pieces of information are available only indirectly in the change history. The version control system is designed to maintain versions of the source code but has no data on key variables like quality, purpose, or difficulty of a change. Second, geographically distributed development leads to distributed data sources as well as the need to distribute the results. The development of the product occurs in more than three locations in the US as well as in more than four other countries. To address these challenges, we developed and implemented techniques for augmenting the change data by estimating the purpose of each change, by estimating change effort drivers, and by constructing and exploring several measures to assess change size, complexity, and developer expertise.

In §II, we describe the specifics of the software product under consideration together with the change management process and the structure of change data. §III describes the measurements we obtained together with some data analyses that we performed. In §IV we illustrate descriptive exploratory tools. Finally, in §V we describe the overall architecture of *SoftChange* designed to address the technical challenges.

[1]In 5ESS terminology, a module is a collection of files stored in the same directory; the files tend to have related functionality.

II. Change management data

Commercial software is rarely static, as it must evolve to satisfy new business needs and to accommodate new hardware or new standards. The following sections investigate evolution of a large software product. For concreteness we describe the change process and data specific to the product. Most other medium to large software development projects use similar processes and collect similar data.

We examined the 5ESS$^{\text{TM}}$, a large switching software project with a rich change history: it spans sixteen years and comprises 100,000,000 lines[2] of source code (in C/C++ and a proprietary state description language) and 100,000,000 lines of header and make files, organized into some 50 major subsystems and 10,000 modules. Any one generic (release) of the system involves some 20,000,000 lines of code.

A. The change process

The changes to the source code follow a well-defined process. New *software releases* or *software updates* are customer deliveries that contain new functionality (features) and fixes or improvements to the code. *Features* (for example, call waiting or credit card billing) are the fundamental design unit by which the system is extended. Very large changes that implement a feature or solve a problem are decomposed by development managers into *initial maintenance requests* (IMRs).

Associated with each IMR are a number of *maintenance requests* (MRs), which are information representing the work to be done to each module. (Thus, an IMR is a problem, while an associated MR is all or part of the solution to the problem.) To perform the changes, a developer "opens" the MR, makes the required modifications to the code, checks whether the changes are satisfactory (in a limited context, i.e., without a full system build), and then submits the MR. Code inspections and integration and system tests follow. An MR is thus the largest unit of change normally performed by a single developer. The supervisor responsible for the IMR distributes the work to the developers.

An editing change to an individual file is embodied in a *delta*: the file is "checked out," edited and then "checked in."

B. Change data

We present a simplified description of the data collected by the Source Code Control System (SCCS) [15], Extended Configuration Management System (ECMS) [16]. In addition to this fairly standard change management data we used some information from a proprietary fault and feature

[2]Numbers are approximate.

tracking system (IMRTS). The structure of the changes is illustrated in Figure 1.

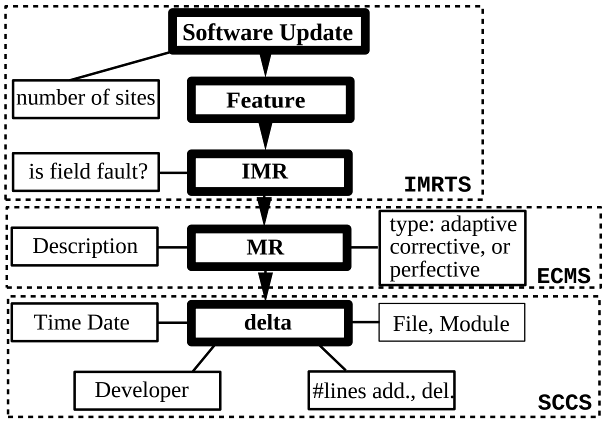


Fig. 1. Change hierarchy and properties.

Changes can be aggregated to several different levels. Coarser levels than deltas, MRs, and IMRs are *features*, which correspond to requirements in traditional models of software development and which typically require hundreds of IMRs, and *software releases or software updates*, which correspond to features and fix IMRs that went into a particular software release or software update. Smaller products typically have less levels of such hierarchy. The information at the IMR and MR level would be merged together, and features, software updates, and releases are often collapsed into a single level in smaller products.

ECMS, like most version control systems, operates over a set of files containing the text lines of source code. An atomic change, or *delta*, to the program text consists of the lines that were deleted and those that were added in order to make the change. Deltas are usually computed by a file differencing algorithm (such as Unix `diff`), invoked by SCCS, which compares an older version of a file with the current version. The data for each delta list the parent MR and the date and time when the change was submitted to the version management system as well as numbers of lines added, deleted, and unmodified by that change. It also keeps the file ID that can be used to find the file name and module in one of the ECMS relations.

ECMS records the following attributes for each change: the file with which it is associated; the date and time the change was "checked in"; and the name and name of the developer who made it. Additionally, the SCCS database records each delta as a tuple including the actual source code that was changed (lines deleted and lines added), name of the developer, MR number (see below), and the date and time of change.

Due to the extensive size of the software, the SCCS and ECMS databases have a separate instance for each subsystem and those instances were located on five different computers. The size of the SCCS database for an average subsystem is 60Mb. The ECMS database averages 120Mb per subsystem. There are around 80 subsystems so that the total volume of change management data is approximately 15 Gb.

III. Analysis tools

Before performing data analyses, we often found it necessary to augment the summary data by constructing a number of change measures. Examples are the size of changes and their complexity, measures of the expertise of the developers who worked on the changes, and a measure of the purpose of the change. After describing these important measurements, we discuss our data analyses, which include identifying factors that drive effort necessary to implement changes, predicting the quality of a change, and measuring the modularity properties of the software. The data preparation part of analysis tools are implemented as *Perl* scripts and modelling tools are implemented in *S* language [17].

A. Obtaining change size, complexity and developer expertise

In addition to the data available directly from SCCS and ECMS, we constructed and explored a number of measures of change size, complexity, and developer expertise.

We found that measuring change properties is essential to understand and analyze software evolution. *SoftChange* computes a set of software change properties that we found useful in one or more applications. The basic properties of the change that are recorded by a simplest change management system include ownership (who made the change), object that is changed (product, subsystem, module, file, and the set of lines), and time when the change was performed. We also estimate additional measures on changes. We group such measures into five classes: size measures, duration measures, complexity measures, expertise measures, and change purpose measures.

The size measures include numbers of lines of code (LOC) added and deleted, and LOC in the files touched by the change. The change size is also measured by the number of sub changes. For example, the size of a feature may be measured in numbers of IMRs, MRs or deltas, and the size of an MR may be measured in number of deltas. Duration measures indicate the temporal interval spanned by the change, for example, difference in time between the last and first delta or between the open and close times of the change.

Change complexity or interaction measures include the total numbers of files, modules, or subsystems touched by a change, or the numbers of developers or organizations involved in the change.

Change expertise measures are based on the average expertise of developers performing the change. Developer expertise is determined by the number of deltas completed by a developer before the change is started.[3] Two modifications of developer expertise measures are also computed: recent expertise, where recent deltas are weighted more heavily than delta performed a long time ago; and subsystem expertise, where only deltas on the subsystems that a change touches are included in calculating developer expertise.

B. Obtaining change purpose

There are three primary driving forces in the evolution of software: *adaptive* changes introduce new functionality, *corrective* changes eliminate faults, and *perfective* changes restructure code in order to improve understanding and simplify future changes (see, e.g., [18], [19]). Quantitative modeling of software evolution must take into account the significant differences in purpose and implementation of the three types of changes (see, e.g., [11], [7], [8]).

Ideally, each delta should be classified as to whether it was adaptive, corrective, or perfective. However, the version management system does not capture this information directly, and instead records short abstracts describing purposes of changes at the MR level. We have used textual analysis of the MR abstracts to impute adaptive, corrective, or perfective labels to the changes [10]. Upon taking out an MR, developers write a short description of the purpose for the change, and we have classified MRs as adaptive, corrective, or perfective depending on which key words appear in these descriptions. The classification scheme was able to tag around 85% of all MRs. In the 5ESS software 5% of MRs were done to implement recommendations of code inspection meetings. Such MRs were easy to identify from their abstracts. Their purposes, however, were difficult to fit into the three categories, since inspection MRs are mostly performed for new feature IMRs, but the inspection recommendations are more likely to involve cosmetic/perfective maintenance (for example, better comments and naming conventions) or fixes of potential errors. Due to those reasons, we classified such MRs into a separate *inspection* class.

A more detailed description of the classification procedure and an assessment of its precision are reported in [10].

C. Other measurements of software change

Some properties of changes are difficult to estimate from the change management data or the estimates have large uncertainty. Such properties should be obtained using surveys to augment estimates from change management data. Since surveys are relatively expensive, only a small subset of changes can be measured. However, if the subset of changes is chosen appropriately, such measures may provide invaluable resource to calibrate, validate, and improve the precision the analysis tools using automat-

[3]Thus, we use past activity and experience to measure expertise.

ically derived change measures. Examples of such direct measurements are given in the two analysis examples below.

The first example uses a change effort survey for changes made by a set of developers. The second example uses root cause analysis on faults reported for all post-release failures from a subset of releases. Such analysis shows which changes in the considered subset were the cause of the post-release failure and which were not. If a random selection of failures for the root cause analysis were used (this is a more traditional way to select a subset of faults to perform root cause analysis on), it would not be capable of identifying changes that did not lead to post-release failure. See also [7] for examples of data obtainable through interviews.

Analysis strategies in §III-D–III-E discuss how such additional measures could improve inference on cost and quality drivers.

D. Estimating effort

A particularly important quantity related to software is the cost of making changes. Therefore, it is of great interest to understand which factors have historically had strong effects on this cost, which we approximate by the amount of time developers spend working on the change.

When performing historical studies of cost necessary to make a change, it is important to study changes at a fine level (MRs in our case). Studying larger units of change, such as features, can make it impossible to sort out the effects of some important factors. For example, features typically contain a mixture of several types of changes, especially both new code and bug fixes, and often also perfective maintenance, so the relative difficulties of the different types of changes are not estimable at the feature level. Also, larger change units can involve multiple developers and distant parts of the code, making it difficult to estimate developer effects or measure, for example, decay of a single subsystem.

Measurements of change effort are not directly recorded by the change management system, so it is necessary to use an algorithm described in [11] and implemented in *SoftChange* to, in effect, divide a developer's monthly effort across all changes worked on in that month. In doing so we make use of a number of measurements on each change that *SoftChange* records. These measurements include the size and type of a change. Both measures are related to the amount of effort required to make the change.

In [11], we applied this procedure to learn that bug fix MRs required 80% more effort than otherwise comparable additions of new code, that effort increases sublinearly with the size of the change, and that one subsystem of the 5ESS™code appeared to be decaying at a rapid rate. We have since applied the effort modeling methodology to the problem of quantifying the labor-saving effect of a development tool in [12].

Finally, to validate the model we surveyed eight developers to rate the effort required to perform thirty of their own MRs done in the recent past. The results indicated that the models' predictions were in line with developers' experiences.

The effort estimation tools provide valuable cost driver data that could be used in planning and in making decisions on how to reduce expenses in software development.

E. Prediction of change quality

Predicting and modeling the possibility of post-release failure has been an active area in software research. One approach, referred to as software reliability, aims to estimate the number of faults remaining in the software in order to generate likelihoods of failure in periods of time; see, for example, [20], [21], [22], [23]. Since post-release failure rates are typically very small, they are modeled for entire releases. In addition, source-code-centric measures are typically used to model the probability of failure, for example in [24], [13], [25], [26], [27], [28]. Historic fault rates for code units can also predict future fault rates; see [29] as well as [8] and its references. As with effort models, the large size of releases confounds a number of factors that contribute to failure. Such factors include experience of individual developers and complexity, type, size and other characteristics of individual changes.

In this analysis example we modeled the probability of failure of Software Updates (SUs), which are very small releases designed for rapid deployment of patches and small new features. Due to the frequency and nature of software updates, the requirements on quality were extremely high - none of the SUs should fail. The development organization consequently performs in-depth root cause analysis, which involves finding the reason of the failure as well as lessons to prevent repeated occurrences of similar failures, on all post-release faults in the changes submitted as software updates. This provided a perfect environment for modeling quality, since we had all IMRs and their properties (extracted by *SoftChange*) that were submitted as SUs and we also had the root cause analyses indicating which IMRs in which SUs caused the failures.

To perform the modeling and prediction ,all change measures described above were computed for IMRs and for SUs using *SoftChange*. Then, generalized linear models [30] to predict the probability of failure of an SU were fit using all available predictors. The models were fit using the oldest 90% of SUs and the predictive power tested on the most recent 10% of the SUs. Only a few of the change measures were needed to obtain the best predictor of SU failure. The failure probability of the SUs increased with the complexity of the software update (measured in numbers of subsystems touched) and with the size of SU measured in number of added lines. The failure probability of the SUs decreases with recent developer expertise.

The same methodology was successful even at the very fine level of IMRs. The failure probability of the IMRs increases with complexity (measured in numbers of modules touched). IMRs written by developers with large amounts of recent expertise were less likely to fail. More details are reported in [31]. *SoftChange* provides interactive Web-based tools so that developers can calculate the probability of failure for a particular software update or IMR.

The quality prediction analyses provided valuable insights into causes of failures of software deliveries. They also led to a number of process changes designed to improve inspection and testing of the most risky IMRs and SUs, as well as to a set of more general improvements in the IMR and SU processes.

IV. Descriptive and presentation tools

The presentation layer is designed to integrate and deliver summary and analysis results to the user. This is implemented using the *Live Documents* (see [32] and [33]) framework, and includes tools to explore and visualize various aspects of the change data. We provide only two simple examples of the presentation layer. The two presentations are *ChangeTrend* (§IV-B), which explores time patterns in the change data, and the *Subsystem Summarizer* (§IV-C), which provides statistical summaries based on change properties for subsystems of the source code.

A. Web delivery by Live Documents

The Web is our choice of presentation medium because of its extensive use and simplicity of deployment. *Live Documents* are World Wide Web-based tools that present documents in a manner that facilitates interactive, reader-guided analysis and visualization of real-time data bases. To accomplish these tasks, the data are represented as interactive tables and histograms accessible through a standard Web browser. In our case, this architecture was used to share the results with other researchers working on a related project as well as with developers and managers responsible for the maintenance of the analyzed software system.

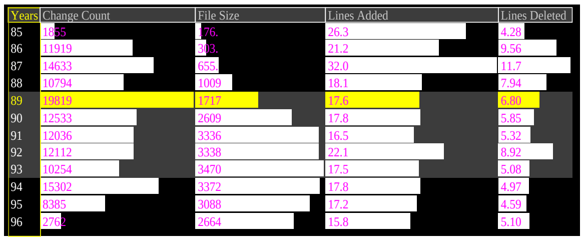
B. Time trends

Time trends of change measures provide valuable information about quality, effort, and interval that could be used in making informed decisions when managing a software project. The *ChangeTrend* part of *SoftChange* is designed to present current status for a number of predetermined trends in change measures. It also allows Web presentation of arbitrary trends (specified via SQL queries) of change measures.

Here we illustrate *ChangeTrend* with the total numbers of changes over time.

Time trends of the change data can be presented in a number of hierarchies (for example, subsystem–module–file, developer–manager–organization, and line–delta–MR) and on different time scales, including hour of the day, day of the week, month of the year, and annual.

Figure 2, for example, shows yearly time trends. From this Figure, we see that the average size of changed files grows until 1993, and then decreases. Numbers of changes have peaks in 1987, 1989, and 1994. Yearly averages of added and deleted lines have peaks in 1985, 1987, and 1992. (There is no peak of deleted lines in 1985 because there was not much code at that time.) The peaks in numbers of changes lag the peaks in numbers of added lines by two years. Since the largest changes are attributed primarily to adaptive maintenance, the peaks in added lines correspond to significant enhancements of functionality. Such enhancements generate a large number of minor problems that are fixed (causing a peak in the corrective maintenance that generates large numbers of changes) during the next major release, which occurs two years later.



Years	Change Count	File Size	Lines Added	Lines Deleted
85	1855	176.	26.3	4.28
86	11919	303.	21.2	9.56
87	14633	655.	32.0	11.7
88	10794	1009	18.1	7.94
89	19819	1717	17.6	6.80
90	12533	2609	17.8	5.85
91	12036	3336	16.5	5.32
92	12112	3338	22.1	8.92
93	10254	3470	17.5	5.08
94	15302	3372	17.8	4.97
95	8385	3088	17.2	4.59
96	2762	2664	15.8	5.10

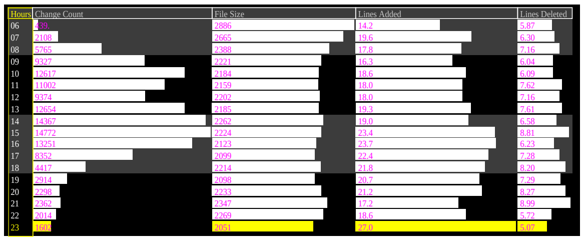
Fig. 2. The yearly submission of changes and their sizes. File sizes grow until 1993 and then decrease. Peaks in adaptive maintenance (large changes) are followed by peaks in corrective maintenance (large numbers of changes).

The interactive tables in Figures 3 and 2 show numeric and textual data, with variable names across the top and values for each observation in row-ordered cells. Three representations of data values are possible, depending on available screen space: as textual numeric digits, as thin bars with lengths proportional to the values, and as a combination of these two, with the digits overplotted on the bars. The rows of the table can be sorted to show relationships among the variables. Scrollbars (not shown) on the left side of the view control the available screen space and scroll the tables.

The numbers of changes exhibit strong trends over smaller time scales as well, including months of the year (due to a yearly project cycle), days of the week, and hours of the day. Figure 3 shows hourly time patterns. (Hours from 12:00 midnight to 5:00 AM are not shown since they contain only small numbers of changes.) There is a clear hourly trend of activity with one peak after lunch and one before,[4] with decreases in activity at night and during lunch time. The column with average numbers of added lines shows that the largest changes are being submitted just before midnight. Further investigation

[4]All the developers work in a single time zone.

revealed that these changes are associated with adaptive maintenance; their timing ensures access to late shifts of the testing laboratory during peak development periods.



Hours	Change Count	File Size	Lines Added	Lines Deleted
06	439.	2886	14.2	5.87
07	2108	2665	19.6	6.30
08	5765	2388	17.8	7.16
09	9327	2221	16.3	6.04
10	12617	2184	18.6	6.09
11	11002	2159	18.0	7.62
12	9374	2202	18.0	7.16
13	12654	2185	19.3	7.61
14	14367	2262	19.0	6.58
15	14772	2224	23.4	8.81
16	13251	2123	23.7	6.23
17	8352	2099	22.4	7.28
18	4417	2214	21.8	8.20
19	2914	2098	20.7	7.29
20	2298	2233	21.2	8.27
21	2362	2347	17.2	8.99
22	2014	2269	18.6	5.72
23	1602	2051	27.0	5.07

Fig. 3. The hourly submission of changes and their sizes. Although most changes are done during working hours, the largest changes occur at 11PM. The first three columns show hours since midnight, numbers of changes, and average size of the file being changed. The fourth through fifth columns show average (over all changes for that hour) numbers of added and deleted lines.

The next section shows how the source code may be characterized by the properties of changes made to the code.

C. Subsystem Summarizer

Here we describe the *Subsystem Summarizer*, a tool that automatically generates an interactive Web page that summarizes important characteristics of code units. It calculates properties of the changes to those code units derived from the change history.

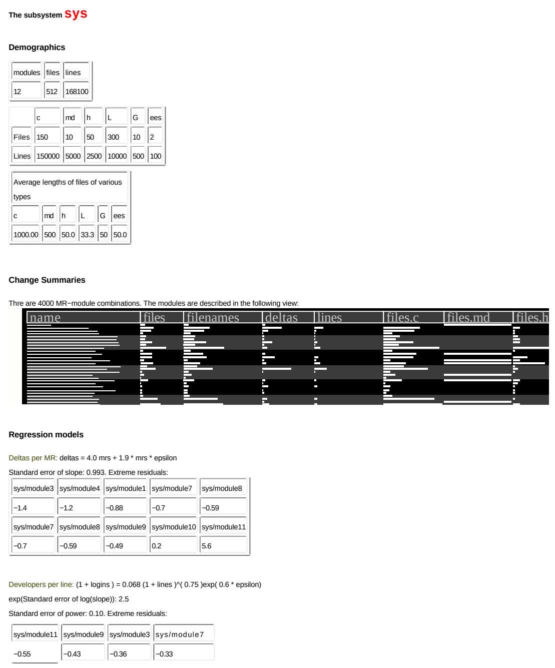
We began study of the software by concentrating on a single subsystem, a code unit both large and tractable. This choice also made it possible to establish strong contacts with the managers and programmers charged with developing the subsystem, leading to improved understanding of the code and the organizational events that affected it [7].

A product of the study of this subsystem was an understanding of what analyses are appropriate for these data. This understanding is embodied in a Subsystem Summarizer, which is now applied automatically to all the subsystems to incorporate the most recent changes.

An abridged version of the output from the summarizer applied to a relatively small subsystem is shown in Figure 4.[5] The summarizer reports demographics of the subsystem, such as the number of modules, files, and lines of code, the latter two classified in terms of the programming languages used in the subsystem. Most of the code in the subsystem depicted in Figure 4 is written in C, and the subsystem also contains files of other types (make files and headers of various types), which tend to be smaller than the C files.

The "Change Summaries" display makes the summary a Live Document (§IV-A), by allowing interactive exploration of the modules and their numbers of files and lines of various types. Since we

[5]With the names of code units and numerical values changed to protect proprietary information.



The subsystem sys

Demographics

modules	files	lines
12	512	168100

	c	md	h	L	G	ees
Files	150	10	50	300	10	2
Lines	150000	5000	2500	10000	500	100

Average lengths of files of various types					
c	md	h	L	G	ees
1000.00	500	50.0	33.3	50	50.0

Change Summaries

Thre are 4000 MR−module combinations. The modules are described in the following view:

Regression models

Deltas per MR: deltas = 4.0 mrs + 1.9 * mrs * epsilon

Standard error of slope: 0.993. Extreme residuals:

sys/module3	sys/module4	sys/module1	sys/module7	sys/module8
−1.4	−1.2	−0.88	−0.7	−0.59
sys/module7	sys/module8	sys/module9	sys/module10	sys/module11
−0.7	−0.59	−0.49	0.2	5.6

Developers per line: (1 + logins) = 0.068 (1 + lines)^(0.75)exp(0.6 * epsilon)

exp(Standard error of log(slope)): 2.5

Standard error of power: 0.10. Extreme residuals:

sys/module11	sys/module9	sys/module3	sys/module7
−0.55	−0.43	−0.36	−0.33

Fig. 4. Extracts from output of the Subsystem Summarizer. The (proprietary) numbers have been altered.

may sort the modules by their values of any of the variables and restrict attention to subsets of the modules, we may ask questions as complex as "Of modules with more than one .h file, how strong is the relationship between number of lines of C code and number of deltas?"

Summarizer output also includes regression relationships among different measurements on the modules and identifies outliers in these relationships. (The regression procedure includes evaluating whether a linear model is appropriate, and running weighted regressions that yield sensible fits even in the presence of variance heterogeneity.) We see that in a typical module, four deltas are required to implement an average MR, and that the number of developers that touch a module increases more slowly than linearly (approximately at the rate of the 0.75 power) with the length of the module. The module `sys/module11` shows up as an outlier in both regression relationships: the MRs affecting it required unusually large numbers of deltas, and the module was relatively long for the number of developers that worked on it. Since deltas are restricted to affect single files, large numbers of deltas per MR may indicate that changes to this module tend to be spread across large numbers of files and are hence difficult to implement. Supporting this hypothesis is the fact that relatively few developers modify this module. If only expert developers are allowed to modify it, this module may be decayed and a good candidate for restructuring.

The summarizer also implements fault potential models described in [8]. Since the numbers of modules and MRs involved in this subsystem are too small for the results to be reliable, these results are not shown here.

V. System architecture

To facilitate extensions and portability, the *SoftChange* system has modular structure consisting of four principal parts (see Figure 5). Access engines described in §V-A extract summary data from the underlying software databases (ECMS and SABLIME; see `http://www.bell-labs.com/project/sablime/`). As was discussed above, the summary engine stores and queries the extracted data, while analysis tools augment and model software data. Finally, the user interface layer provides visualization and navigation aids.

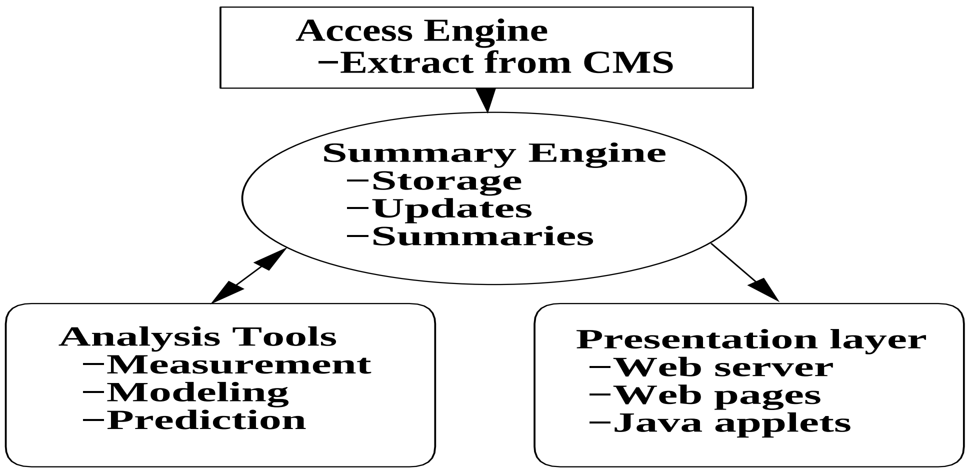


Fig. 5. The four main parts of the system.

The access engine runs periodically to update the primary tables using the most recent changes in change management system. The summary engine then augments the data, reruns the analysis (using analysis tools), and updates the SQL database. The user interface layer includes the Web server, Java applets, and a set of Web pages.

A. Access engine

The role of the access engines is to obtain five primary tables from the configuration management data: logical changes, atomic changes (deltas), files, developers, and faults and features. The access engine is implemented via a number of *Shell* and *Perl* scripts. The inputs specify the location of the SABLIME, SCCS, and fault/feature databases (some organizations record faults and features in the SABLIME database). The outputs are the five primary tables. In the first stage of this process the script is launched (via rsh) on the remote computers storing the SCCS, ECMS, and other databases. To minimize the load on those critical machines a *Shell* script simply packs the remote data into tar files and copies them to the server running the *SoftChange* system. In the second stage of this process, the SCCS files and selected ECMS or SABLIME relations[6] are processed to generate four of the tables, while the fault and feature database is processed into the fifth table.

Periodically, we merge updates to the change history with existing summaries. The new MRs and deltas are extracted, classified and merged with existing databases.

The access engine also performs some data cleanup activities. The version control system keeps track of releases of software by creating *branch deltas*, which can be thought of as assignment of a common label to a particular set of file/version pairs. Such deltas, which do not add or delete any source code, reflect the release history of software which can be obtained directly from configuration management data or from other sources. Consequently the branch deltas are not included in the atomic change table.

To facilitate MR classification (see §III-B), the access engine processes MR abstracts, which are textual descriptions of the purposes of the MRs written by the developer. It converts text to lower case, replaces all delimiters with a single space, removes non-alphanumeric symbols, and uses WordNet [34] to remove the suffix of each word.

Description of the primary tables

The logical change or MR relation contains a change identifier, open date, a short abstract describing the change, and the name of the developer. The atomic change (delta) relation contains the version number, file ID, date and time the atomic change was submitted, numbers of lines added, deleted, and unmodified by the change, name of the developer who implemented the atomic change, and the reference to the logical change. The file relation maps file identifier to subsystem, module, and file name. The developer relation maps developer login to the full name and organization.

The MR table lists MR, MR abstract, estimated MR purpose, the release in which the MR was incorporated, login and full name of the developer, geographic location of the developer, and the time stamp when MR was opened. Since a developer's full name may be spelled in different ways and sometimes changes, we keep the full name associated with each MR.

The delta table contains file identifier, date, time, version, lines added, deleted, and unchanged, developer login, and MR identifier.

The developer table lists logins and all possible spellings of full names of the developers. The file table lists file IDs paired with the file and module names.

The IMR tracking system records a number of fields for each IMR. *SoftChange* extracts fields containing release number, software update number, feature number, an indicator describing at what

[6]The MR and ORG relations for MR abstracts and developer names, and the GM relation to map file IDs to file names.

point in the development process the IMR originated (coding, testing, post release), and an indicator of whether the IMR was a bug fix or a new feature.

B. Summary engine

The summary engine stores the data and provides summaries and subsets of the data, for example, a list of changes over the last month or a list of developers who added the most lines in each module. The tables are stored in flat files and in the SQL database. The summary engine is also responsible for updating all tables with the newest data from the configuration management system. It is essential to update the tables incrementally because complete retrieval of summaries from 5ESS version control data takes several days.

The additional tables are precomputed to speed certain queries and to store additional values computed using the analysis tools. The three most important additional values are change purpose, change effort, and developer expertise.

The information extracted from SCCS and ECMS databases are linked to provide for faster access and simpler analysis. Information on MR type is propagated to the delta table, while number of deltas, time of first and last delta, and a list of modules and files touched are propagated to the MR table.

The additional tables are computed at delta, MR, IMR, feature, and release levels. Delta summaries include the following information.[7]

[7]MR purposes include bug, new, cleanup, inspection rework, and unknown; for more details see Section III-B. Examples of geographic locations are Indian Hill in Illinois, Hilversum in the Netherlands, and Malmesbury, United Kingdom.

file ID
file version
time stamp of delta
numbers of lines added
numbers of lines deleted
numbers of lines unchanged
login of the developer
MR number
subsystem
module
file
file extension
IMR number
MR purpose
MR release information
login used to open the MR
geographic location of the MR
timestamp when MR was open
list of IMR release numbers (release)
list of IMR bug/new classification
pipe separated list of features
when IMR originated (coding/testing/field)
software updates the IMR was included in

The information at the MR, IMR, and feature levels also includes lists of files, modules, and subsystems touched by the change, a list of developers participating in the change, and the time of first and last delta.

The developer expertise summary table lists the total number of deltas completed by each login up to a particular month on a particular subsystem. The fields are login, month, subsystem, number of delta done in that month on that subsystem.

The additional tables are imported into a relational database (Oracle on a Windows NT PC) to enable easy calculation of data roll-ups over subsystems, modules, developers, MRs, IMRs, features, and releases.

VI. Discussion

The main contribution of the paper is to define and construct essential measures of software changes, to automate the measurement process, and to integrate those measurements with analysis and presentation tools. Change measures include change complexity, size, purpose, and developer expertise. This basic set is extracted and estimated from a variety of version control databases and is augmented by derived measures that predict change quality and cost. The *SoftChange* system provides a Web-based user interface to display trends in change measures and access to analysis tools and summary results.

The *SoftChange* system demonstrates that version control databases are invaluable resources for understanding software evolution and the software production process. It also points out the importance of studying individual source code changes as opposed to differences between entire versions of the software or study of the static source code.

To be able to study software changes it was essential to handle large and complex data sets. The volume, complexity, and lack of structure of software change data overwhelm standard statistical analysis tools. We have developed special purpose analysis tools, tuned to software engineering data analysis, that both scale to the data volumes associated with large-scale systems and cope with error patterns we encountered when analyzing this data. The tools include new visualization methods to understand software changes.

It is important to point out that our results, intuition and insights come from investigating a software product that has been in service for many years. Not only is this product of very high quality, but the production process can be an example to many other software products, as we are learning by applying the *SoftChange* system to other products.

The *SoftChange* system codifies the knowledge collected through many years of study of a large and commercially successful software product. Because this knowledge is embedded in a portable system it can be (and currently is being) transferred to other software products.

References

[1] L. A. Belady and M. M. Lehman, "Programming system dynamics, or the meta-dynamics of systems in maintenance and growth," Tech. Rep., IBM Thomas J. Watson Research Center, 1971.

[2] M. H. Halstead, *Elements of Software Science*, Elsevier North-Holland, 1977.

[3] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308–320, 1976.

[4] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Software Eng.*, vol. 20, no. 6, pp. 476–493, 1994.

[5] A. J. Albrecht and E. G. John Jr., "Software function, source lines of code, and development effort prediction: a software science validation," *IEEE Trans. Software Eng.*, vol. 9, no. 6, pp. 639–648, 1983.

[6] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does code decay? Assessing the evidence from change management data," *IEEE Transactions on Software Engineering*, 1999, To appear.

[7] N. Staudenmayer, T. L. Graves, J. S. Marron, A. Mockus, H. Siy, L. G. Votta, and D. E. Perry, "Adapting to a new environment: how a legacy software organization copes with volatility and change," in *5th International Product Development Conference*, Como, Italy, 1998.

[8] T. L. Graves, A. F. Karr, J. S. Marron, and H. P. Siy, "Predicting fault incidence using software change history," *IEEE Transactions on Software Engineering*, 1999, To appear.

[9] D. E. Perry, H. P. Siy, and L. G. Votta, "Parallel changes in large scale software development: An observational case study," in *Proceedings of the 1998 International Conference on Software Engineering*, Kyoto, Japan, April 1998.

[10] A. Mockus and L. G. Votta, "Identifying reasons for software changes using historic databases," Tech. Rep., Bell Laboratories, 1997.

[11] T. L. Graves and A. Mockus, "Inferring change effort from configuration management databases," in *Metrics 98: Fifth International Symposium on Software Metrics*, Bethesda, Maryland, November 1998, pp. 267–273.

[12] D. Atkins, T. Ball, T. Graves, and A. Mockus, "Using version control data to evaluate the effectiveness of software tools," in *1999 International Conference on Software Engineering*. 1999, ACM Press.

[13] V. R. Basili and B. T. Perricone, "Software errors and complexity: An empirical investigation," *Communications of the ACM*, vol. 27, no. 1, pp. 42–52, January 1984.

[14] V. R. Basili and D. M. Weiss, "A methodology for collecting valid software engineering data," *IEEE Transactions on Software Engineering*, vol. 10, no. 6, pp. 728–737, 1984.

[15] M.J. Rochkind, "The source code control system," *IEEE Trans. on Software Engineering*, vol. 1, no. 4, pp. 364–370, 1975.

[16] A. K. Midha, "Software configuration management for the 21st century," *Bell Labs Technical Journal*, vol. 2, no. 1, Winter 1997.

[17] R. A. Becker, J. M. Chambers, and A. R. Wilks, *The new S language*, Wadsworth&Brooks/Cole, 1988.

[18] E. B. Swanson, "The dimensions of maintenance," in *Proc. 2nd Conf. on Software Engineering*, San Francisco, 1976, pp. 492–497.

[19] K. H. An, D. A. Gustafson, and A. C. Melton, "A model for software maintenance," in *Proceedings of the Conference in Software Maintenance*, Austin, Texas, September 1987, pp. 57–62.

[20] J. D. Musa, A. Iannino, and K. Okumoto, *Software Reliability*, McGraw-Hill Publishing Co., 1990.

[21] S. N. Mohanty, "Models and measurements for quality assessment of software," *ACM Computing Surveys*, vol. 11, no. 3, pp. 251–275, September 1979.

[22] S. G. Eick, C. R. Loader, M. D. Long, L. G. Votta, and S. VanderWiel, "Estimating software fault content before coding," in *Proceedings of the 14th International Conference on Software Engineering*, Melbourne, Australia, May 1992, pp. 59–65.

[23] D. A. Christenson and S. T. Huang, "Estimating the fault content of software using the fix-on-fix model," *Bell Labs Technical Journal*, vol. 1, no. 1, pp. 130–137, Summer 1996.

[24] K. H. An, D. A. Gustafson, and A. C. Melton, "A model for software maintenance," in *Proceedings of the Conference on Software Maintenance, Austin, Texas*. September 1987, pp. 57–62, IEEE Computer Society Press.

[25] N. F. Schneidewind and H.-M. Hoffman, "An experiment in software error data collection and analysis," *IEEE Trans. on Software Engineering*, vol. SE-5, no. 3, pp. 276–286, May 1979.

[26] N. Ohlsson and H. Alberg, "Predicting fault-prone software modules in telephone switches," *IEEE Trans. on Software Engineering*, vol. 22, no. 12, pp. 886–894, December 1996.

[27] V. Y. Shen, T.-J. Yu, S. M. Thebaut, and L. R. Paulsen, "Identifying error-prone software–an empirical study," *IEEE Trans. on Software Engineering*, vol. SE-11, no. 4, pp. 317–324, April 1985.

[28] J. C. Munson and T. M. Khoshgoftaar, "Regression modelling of software quality: Empirical investigation," *Information and Software Technology*, pp. 106–114, 1990.

[29] T. J. Yu, V. Y. Shen, and H. E. Dunsmore, "An analysis of several software defect models," *IEEE Trans. on Software Engineering*, vol. 14, no. 9, pp. 1261–1270, September 1988.

[30] P. McCullagh and J. A. Nelder, *Generalized Linear Models, 2nd ed.*, Chapman and Hall, New York, 1989.

[31] A. Mockus, H. Siy, J. Y. Chen, T. Graves, and T. Sundresh, "Role of change size, complexity, and developer expertise in predicting the quality of a software update," Tech. Rep., Bell Laboratories, 1998.

[32] S. G. Eick, A. Mockus, T. L. Graves, and A. F. Karr, "A web laboratory for software data analysis," *World Wide Web*, vol. 1, no. 2, pp. 55–60, 1998, See also http://www.baltzer.nl/www/1-2.html.

[33] A. Mockus, S. Hibino, and T. Graves, "Flexible information visualization components for authoring WWW Live Documents," Tech. Rep., Bell Laboratories, 1999.

[34] R. Beckwith and G. A. Miller, "Implementing a lexical network," *International Journal of Lexicography*, vol. 3, no. 4, pp. 302–312, 1990.